TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky a mezioborových studií

Studijní program: B2612 Elektrotechnika a informatika
Studijní obor: Informatika a logistika

Jádro procesoru Z80 firmy ZiLOG v FPGA obvodu

ZiLOG's Z80 CPU core in FPGA circuit

Bakalářská práce

Autor: **Jiří Blažek**
Vedoucí práce: Ing. Zbyněk Mader, Ph.D.
Konzultant: Ing. Martin Rozkovec

V Liberci 21.5.2010

Prohlášení

Byl jsem seznámen s tím, že na mou bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb., o právu autorském, zejména § 60 – školní dílo.

Beru na vědomí, že Technická univerzita v Liberci (TUL) nezasahuje do mých autorských práv užitím mé bakalářské práce pro vnitřní potřebu TUL.

Užiji-li bakalářskou práci nebo poskytnu-li licenci k jejímu využití, jsem si vědom povinnosti informovat o této skutečnosti TUL; v tomto případě má TUL právo ode mne požadovat úhradu nákladů, které vynaložila na vytvoření díla, až do jejich skutečné výše.

Bakalářskou práci jsem vypracoval samostatně s použitím uvedené literatury a na základě konzultací s vedoucím bakalářské práce a konzultantem.

Datum

Podpis

Abstrakt

Cílem této bakalářské práce je implementovat jádro osmibitového procesoru Z80 na desce FPGA Spartan-3.
Význam práce spočívá hlavně v praktickém procvičení si reverzního inženýrství a programování (konfigurování) FPGA obvodů za použití jazyka VHDL, dává si však také za cíl poukázat na stále rostoucí oblibu FPGA obvodů a neutuchající slávu kultovního procesoru Z80.
Implementace pokrývá zavedení registrů, aritmeticko-logické jednotky, externí paměti a samotného jádra centrální řídící jednotky (CPU), včetně zapracování cca. ¼ instrukční sady.
Demonstrační program je napsán v assembleru a jeho funkčnost je demonstrována v přiložených záznamech test benchů.

Klíčová slova: Z80, CPU, FPGA, VHDL

Abstract

The purpose of this bachelor thesis is to implement a Z80 eight-bit CPU's core in a FPGA circuit Spartan-3.
The significance of this work consists mostly within the practising of reverse engineering and in programming (configuring) FPGA circuits using VHDL language, while also highlighting the rising popularity of FPGAs and the cult of the famous Z80 processor.
The implementation covers the introduction of registers, ALU, external memory and – of course – CPU core logic, including approx. ¼ of the instruction set.
The test program is written in assembler and its functionality is being demonstrated by the attached test benches.

Key words: Z80, CPU, FPGA, VHDL.

Obsah

Seznam zkratek

ALU	Arithmetic and Logic Unit – Aritmeticko-logická jednotka
CLK	Clock – Řídící hodinový signál
CPU	Central Processing Unit – Centrální řídící jednotka
DH	Doběžná hrana
FPGA	Field Programmable Gate Array – Programovatelný obvod
HDL	Hardware Definition Language – Hardware popisující jazyk
IS	Instruction Set – Instrukční sada
LUT	LookUp Table – Vyhledávací tabulka
M(n)	M-cyklus č.(n)
MEM	Memory – Paměť
NH	Náběžná hrana
T(n)	T-stav č.(n)
VHDL	Very high speed integrated circuits HDL – jazyk HDL

Úvod

Tato práce si dala za cíl implementovat jádro Z80 CPU na FPGA Spartan-3.

Podařilo se mi to, deska však nebyla schopna pojmout celý konfigurační soubor (tj. na FPGA Spartan-3 je mnou vytvořený VHDL kód pro svoji rozsáhlost nesyntetizovatelný). Práci tak odevzdávám ve formě zdrojových kódů a ukázek behaviorálních testů.

Mnou vytvořený model zvládá kus instrukční sady, práci s registry, ALU a paměť, a tuto schopnost dokládá na jednoduchém testovacím programu v assembleru. Na přiloženém médiu je možné nahlédnout do postupů, jež jsem použil, popř. si i zopakovat test, uvedený ve 3. kapitole tohoto dokumentu, nebo si vytvořit další vlastní testy.

Rád bych na tomto místě poděkoval svojí ženě Martině a babičce z Potoka, které mi po celou dobu práce na tomto zajímavém projektu vytvářely vskutku ideální podmínky.

1 Procesor Z80

Procesor Z80 byl firmou ZiLOG (založenou v roce 1974 Ralphem Ungermannem) poprvé představen v roce 1976.

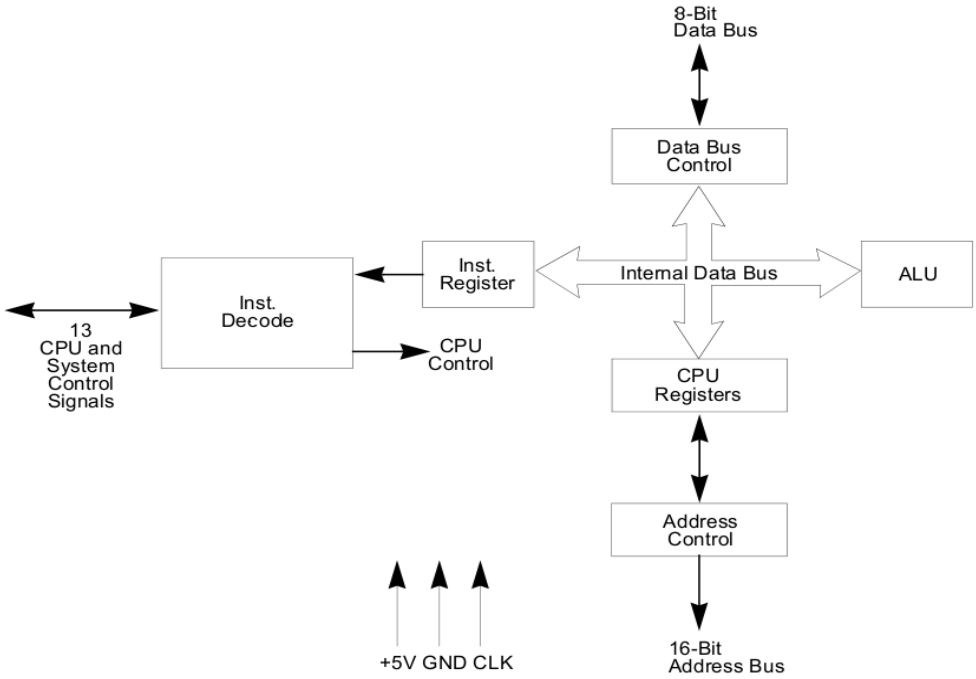
Jeho architektura vychází z procesoru 8080 firmy Intel, což není náhoda, protože spoluautor Z80, Federico Faggin, se v Intelu podílel na vývoji 4bitového mikroprocesoru 4004, předchůdci prvního osmibitu 8008 s následníkem v podobě právě 8080.

Mimo jiné i díky své kompatibilitě s 8080 byl čip Z80 masivně používán v 70. a 80. letech 20. století (bez dalších úprav mohl sloužit jako základ pro operační systém CP/M, původně určený pro 8080), jeho obliba však přetrvává dodnes. Vyhledáván je především pro svoji jednoduchost a spolehlivost.

Jako hlavního zástupce vlny procesorů rodiny Z80 jmenujme slavný mikropočítač ZX Spectrum, jehož deriváty jsou živé dodnes.

1.1 Architektura

Z80 CPU je jednoduchý 8-bitový mikroprocesor, jehož schema je následující:



Obr. 1: Schema Z80 CPU [5]

Z80 CPU je napájen napětím +5V a řízen hodinovým signálem s frekvencí

6-20 MHz. Obsahuje sadu programátorsky přístupných registrů, z nichž některé mohou být použity jako 8- i 16-bitové. Pro 8-bitové aritmetické a logické operace je Z80 CPU vybaven aritmeticko-logickou jednotkou (ALU), s okolím komunikuje za použití čtyřiceti vstupně-výstupních PINů.

1.2 Základní stavební kameny

1.2.1 Oscilátor

Veškerý provoz Z80 CPU je řízen hodinovým signálem CLK (viz obr. 1) s frekvencí 6-20 MHz [5].

Jak je podrobněji rozebráno dále, každá instrukce sestává z aspoň 1 M-cyklu, přičemž každý takový M-cyklus má alespoň 3 T-stavy (M1 cyklus má však vždy alespoň 4 T-stavy).

Protože každé tiknutí hodin odpovídá jednomu T-stavu, můžeme dobu potřebnou k vykonání konkrétní instrukce spočítat takto:

$$T_{instr} = \frac{\text{počet T-stavů}}{\text{frekvence}}$$

1.2.2 Registry

Z80 CPU disponuje celkem 208 bity adresního prostoru, určeného pro registry. Registry jsou 8- a 16-bitové a můžeme je rozdělit do následujících kategorií:

Hlavní registry

8-bitové registry **B**, **C**, **D**, **E**, **H** a **L** lze využít k uchování libovolných informací. Lze na ně pohlížet jako na šest 8-bitových nebo jako na tři 16-bitová úložiště. Jejich využití je libovolné, nicméně z instrukční sady (IS) vyplývá, že některé instrukce (např. LDIR) využívají dvojice registrů primárně takto:

BC – 16-bit byte counter

DE – 16-bit destination register

HL – 16-bit address register (H = high, L = low)

Takto používané dvouregistry jsou v notaci big endian, tj. adresa je v nich uložena následovně:

VYŠŠÍ BYTE | NIŽŠÍ BYTE.

Např. adresa 1023 je reprezentována jako 03FFh, tedy H = 03h a L = FFh.

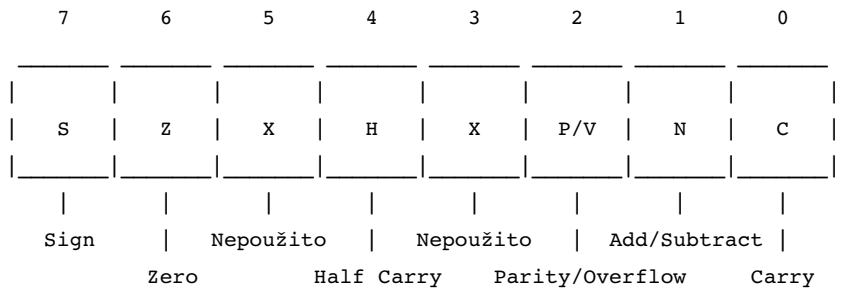
Mezi hlavní registry počítáme také ještě další dva samostatné 8-bitové registry A (Accumulator) a F (Flag register), jež se však nedají spojovat do 16-bitového dvouregistru:

A – Akumulátor

- se používá hlavně pro uchovávání výsledků 8-bitových operací v ALU
- lze do něj volně zapisovat (např. instrukcí `LD A,FFh`) i z něj číst

F – Registr příznaků

- se používá pro uchovávání 6 bitové informace o stavu CPU (z toho 4 bity lze samostatně testovat)
- příznaky jsou nastavovány v ALU (Carry také instrukcemi `SCF` a `CCF`)



Obr. 2: Struktura registru příznaků F

Tyto příznaky mají následující význam:

S (Sign)

7. bit výsledku operace (ukládaného do akumulátoru)

Z (Zero)

je výsledek operace v ALU = 0?

H (Half Carry)

příznak přetečení ze 3. na 4. bit při sčítání nebo podtečení na 4.bitu při odčítání

P/V (Parity/Overflow)

výsledek je sudý nebo větší než platný rozsah (závisí na typu operace)

N (Add/Subtract)

při operaci sčítání nastaven na 0, při odčítání na 1

C (Carry)

příznak přetečení/podtečení výsledku nebo přenášený bit při

rotování a shiftu (závisí na typu operace), lze i programátorsky nastavit instrukcemi `SCF` a `CCF`.

Existují 2 rovnocenné sady hlavních registrů: hlavní (A, F, B, C, D, E, H, L) a alternativní (A', F', B', C', D', E', H', L').

Alternativní sada existuje kvůli rychlému přepínání kontextu při zpracování přerušení – aktuální obsah registrů není nutno nikam ukládat, stačí jen přepnout ukazatel na druhou sadu registrů a po dokončení obslužné rutiny vrátit ukazatel do původního stavu. Procesor však při svém běhu nerozlišuje, která sada je hlavní a která alternativní.

Speciální registry

Speciální registry slouží k uchovávání adres (16-bitové registry PC a SP), adresních bází (16-bitové registry IX a IY) nebo částí adres (8-bitové registry I a R).

PC – Program counter

16-bitový registr PC obsahuje adresu instrukce, která bude načtena z paměti. Obsah registru je po každém načtení instrukce automaticky zvýšen o 1.

Obsah registru lze také měnit instrukcemi `JUMP`, `CALL` nebo `RETURN`.

SP – Stack pointer

16-bitový registr SP obsahuje adresu vrcholu LIFO zásobníku (stacku), umístěného kdekoliv v paměti.

Adresu v SP lze přímo nastavit instrukcemi `LD SP,{HL|IX|IY}` nebo instrukcemi `PUSH` a `POP`, které (krom výměny dat mezi 16-bitovým registrem a vrcholem stacku) tuto adresu v SP automaticky navyšují resp. ponižují o 2.

Registry **IX** a **IY**

16-bitové registry IX a IY se používají jako báze adresy při tzv. indexovém adresování paměti, kdy výsledná adresa = (IX resp. IY) + d, $d \in \langle -128 ; +127 \rangle$.

Obsah registrů IX a IY lze libovolně měnit.

Registr **I**

8-bitový registr I (Interrupt vector) obsahuje horních 8 bitů adresy, na které leží obslužný kód přerušení (dolních 8 bitů adresy dodává zařízení, které přerušení

vyvolalo).
Obsah registru I lze změnit instrukcí `LD I,A`.

Registr **R**
8-bitový registr R (Refresh counter) obsahuje dolních 8 bitů adresy, která je během dekódování instrukce (2. polovina M1 cyklu) použita k občerstvení dynamické paměti RAM (horních 8 bitů adresy pochází z registru I) .
Obsah registru R je po každém načtení instrukce zvýšen o 1, přičemž nejvyšší bit není v čítači zohledněn (jsou měněny bity 0 – 6).
Obsah registru R lze také změnit instrukcí `LD R,A`.

1.2.3 ALU

Aritmeticko-logická jednotka Z80 CPU provádí následující 8-bitové aritmetické a logické operace se vstupními daty, umístěnými v interním datovém kanálu:

`ADD, ADC, SUB, SBC`
: do/z akumulátoru přičte/odečte operand [a Carry flag]

`AND, OR, XOR`
: logický součin/součet/exklusivníOR akumulátoru a operandu

`CP`
: porovná akumulátor s operandem

`INC, DEC`
: zvýší/sníží obsah na dané adrese o 1

`RR[A], RRC[A], RL[A], RLC[A]`
: rotuje obsah na dané adrese o 1 bit doprava/doleva [s použitím Carry flagu]

`SLA, SRA, SRL`
: aritmetický posun obsahu na dané adrese o 1 bit doprava/doleva s použitím Carry flagu

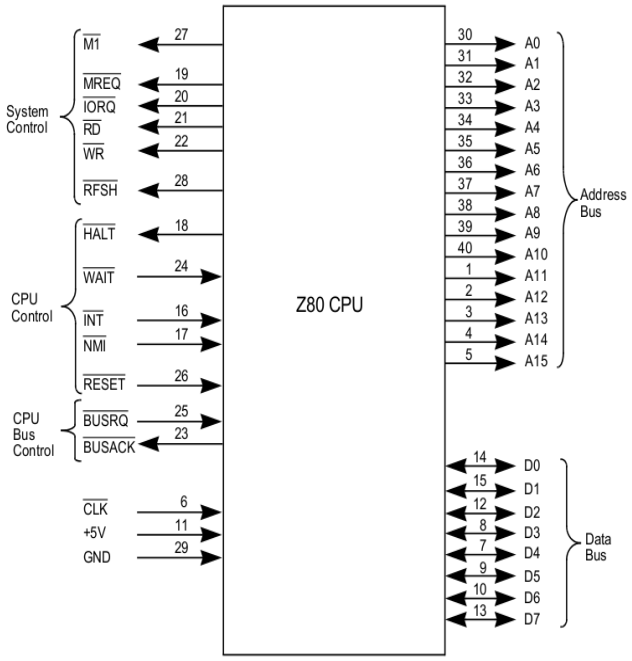
`RLD, RRD`
: 4-bitová BCD rotace doleva/doprava mezi akumulátorem a bytem na adrese uvedené v dvouregistru HL

`SET, RES, BIT`
: nastaví bit na 1 resp. 0 / testuje bit

Všechny tyto operace ovlivňují obsah registru příznaků F.

1.3 I/O PINy

Se svým okolím komunikuje Z80 CPU těmito čtyřiceti I/O PINy:



Obr. 3: Z80 CPU I/O PINy [5]

Tyto vstupní/výstupní signály lze rozdělit do následujících 5 kategorií podle významu (vedle toho však nesmíme zapomenout také na přívod napětí, uzemnění a na synchronizační hodinový signál):

1.3.1 System control

Těchto 6 výstupních signálů řídí činnost paměti a periferií.

Pozor: signály jsou negované (tzn. 0 = aktivní, 1 = neaktivní)!

M1

Oznamuje svému okolí, že CPU se nachází v první polovině M1 cyklu, tedy ve fázi načítání instrukce.

MREQ

Požadavek na přístup do paměti. Tento třístavový signál oznamuje, že address bus obsahuje platnou adresu a pamětí je chápán jako povel k zahájení čtení nebo zápisu (v závislosti na stavu signálů RD a WR).

IORQ

Požadavek na přístup k I/O portu. Tento třístavový signál oznamuje, že dolní

polovina address busu obsahuje platnou adresu; v momentě skenování přerušení je signál využit jako oznámení o možnosti umístit na data bus vektor odpovědi na přerušení.

RD

Tento třístavový signál oznamuje požadavek na čtení z paměti nebo periferie.

WR

Tento třístavový signál oznamuje, že data bus obsahuje platná data, která mají být zapsána do paměti nebo periferie.

RFSH

Tento signál oznamuje, že address bus obsahuje adresu, která může být použita k aktualizaci dynamické paměti.

1.3.2 CPU control

Tyto 4 vstupní signály řídí činnost Z80 CPU, jeden výstupní signál indikuje zastavení systému.

Pozor: signály jsou negované (tzn. 0 = aktivní, 1 = neaktivní)!

HALT

Tento výstupní signál oznamuje, že CPU provedla instrukci `HALT`, a že do příchodu přerušení bude vykonávat instrukce `NOP` (No OPeration).

WAIT

Tento vstupní signál je oznámením paměti nebo periferie, že data nemohou být včas přenesena na data bus. V případě, že je ve fázi čtení, vykonává Z80 CPU instrukce `NOP`, dokud signál WAIT znovu nepřejde do stavu 1.

Tímto způsobem je zajištěno, že Z80 CPU bude zaručeně komunikovat i s velmi pomalými zařízeními.

INT

Požadavek periferie na maskovatelné přerušení. Signál je rozeznáván na konci každé instrukce, k samotnému zpracování však dochází pouze pokud je přerušení povoleno (IFF = 1).

Samotné zpracování přerušení závisí na nastaveném módu přerušení (Mode 0

vede k vykonání instrukce dodané periferií, Mode 1 vede k volání rutiny na adrese 0038H a Mode 2 vede k nepřímému volání rutiny kdekoliv v paměti) [5].

NMI

Požadavek periferie na nemaskovatelné přerušení. Signál je rozeznáván na konci každé instrukce, má vyšší prioritu než INT, a bez ohledu na stav IFF donutí Z80 CPU k vykonání rutiny na adrese 0066h.

1.3.3 CPU Bus control

Tyto 2 signály slouží k realizaci přímého přístupu externích zařízení k prostředkům ovládaným Z80 CPU (DMA – Direct Memory Access).
Pozor: signály jsou negované (tzn. 0 = aktivní, 1 = neaktivní)!

BUSRQ

Tento vstupní signál má vyšší prioritu než NMI a je rozeznáván na konci každého M-cyklu. Je požadavkem externího zařízení na přechod address busu, data busu a třístavových System control signálů MREQ, IORQ, RD a WR do vysokoimpedančního stavu.

BUSACK

Tímto výstupním signálem oznamuje Z80 CPU externímu zařízení, že address bus, data bus a System control signály MREQ, IORQ, RD a WR přešly do vysokoimpedančního stavu a zařízení jsou tak připravena k použití.

1.3.4 Address bus

Těchto 16 třístavových výstupních signálů nese informaci o adrese, se kterou má pracovat paměť nebo periferie.
Piny jsou číslovány A15-A0 (od nejvýznamnějšího k nejméně významnému), adresa je zapsána v notaci big endian.
Celkový rozsah hodnot je 0000h – FFFFh.

1.3.5 Data bus

Těchto 8 třístavových obousměrných signálů nese informaci o datech umístěných v paměti resp. periferii na adrese uvedené v address busu.
Piny jsou číslovány D7-D0 (od nejvýznamnějšího k nejméně významnému).

1.4 Okolní svět CPU

Dosud jsme si popsali, jak vypadá jádro Z80 CPU uvnitř. Procesor však nezmůže nic bez okolních zařízení, která mu umožňují pracovat s informacemi a komunikovat s uživatelem.

1.4.1 Paměť

Z80 CPU používá dva druhy pamětí: paměť pouze pro čtení (ROM) a paměť pro čtení a zápis (RAM).
Námi uvažovaný model ZX Spectrum 64k obsahuje paměti těchto velikostí:

ROM	16kB	(2^{14} bytů)	(0000h – 3FFFh)
RAM	48kB	($2^{16} – 2^{14}$ bytů)	(4000h – FFFFh)

Úhrnná velikost paměťového prostoru je tedy 64 kilobytů (2^{16} bytů), odkud také vyplývá 16-bitová šíře address busu (A15-A0).

Instrukce Z80 CPU jsou uloženy jak v paměti ROM (např. obslužné rutiny přerušení, systém Basic, atd.), tak v paměti RAM (uživatelem definované instrukce a data). Adresa místa v paměti, ze kterého bude načtena příští instrukce nebo její část, je uvedena v registru PC (viz kap. 1.2.2).

Součástí RAM je také VideoRAM, jejíž velikost je 6912 bytů a nachází se ve spodní části RAM na adresách 4000h – 5AFFh. Paměť VideoRAM je určena pro vykreslování grafických informací, jež jsou následně přeneseny na zobrazovací zařízení zákaznickým obvodem ULA (viz dále).

1.4.2 Periferie

Periferiemi rozumíme všechna taková zařízení, která komunikují s CPU prostřednictvím 8-bitového I/O portu.
Pro komunikaci s CPU vyvolává takové zařízení nemaskovatelné resp. maskovatelné přerušení (tj. nastaví signál NMI resp. INT (viz kap. 1.3.2)).

1.4.3 ULA

ULA je zákaznický obvod, který zprostředkovává komunikaci mezi CPU, pamětí, periferiemi a uživatelem.

Obsluhuje mj. 5 vstupních signálů z klávesnice, 3 výstupní video signály, 8 data signálů a další.

Pro účely implementace jádra Z80 CPU na FPGA není implementace ULA nezbytná, navíc svojí náročností překračuje rámec této bakalářské práce.

1.5 Instrukční sada

Instrukční sada Z80 čítá více než 240 instrukcí, které se dají rozdělit do následujících kategorií [5]:

Load and exchange

přenosy 8- i 16-bitových dat mezi registry nebo mezi registry a pamětí/periferií (např. `LD A,(HL)` nebo `PUSH DE`) a výměny obsahů registrů (např. `EX DE,HL` nebo `EX (SP),IY`)

Blokové vyhledávání a přesuny

např. `LDD` nebo `CPIR`

Aritmetické a logické operace

např. `OR (HL)` nebo `DEC IX`

Rotování a posuny

např. `RRCA` nebo `SLA (IX+5)`

Manipulace s bity (Set, Reset, Test)

např. `SET 2,D` nebo `RES 7,(HL)`

Jump, Call, Return

např. `JP (IY)` nebo `RETI`

Input/Output

např. `INI` nebo `OUTD`

CPU control

např. `HALT`

1.5.1 Časování

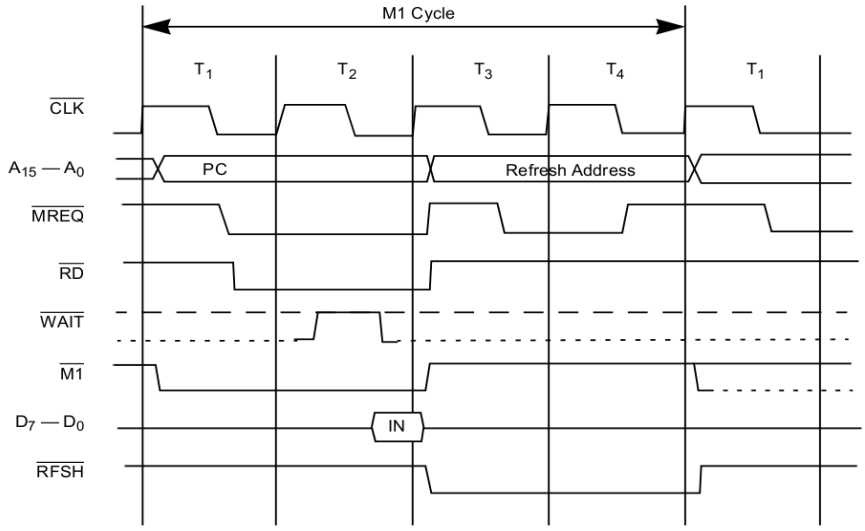
Jak již bylo zmíněno v kap. 1.2.1, je Z80 CPU řízen vstupním signálem CLK. Každému tiknutí hodin pak odpovídá jeden T-stav, jednotlivé T-stavy se skládají do M-cyklů.

Každá instrukce má specifikován přesně určený počet M-cyklů a jejich T-stavů, které vycházejí z počtů cyklů jednotlivých kroků, z nichž se instrukce skládá.

Uveďme si nyní ukázky tří nejzákladnějších kroků:

Načtení a zpracování instrukce

Načtení instrukce a její dekódování je vždy prováděno v M1 cyklu a má pevně daný průběh, uvedený na obrázku:



Obr. 4: Průběh načtení a zpracování instrukce [5]

Při náběžné hraně T1 stavu je na address bus vložen obsah registru PC. Při doběžné hraně T1 stavu jsou již signály A15-A0 ustáleny (adresa je platná), můžeme tedy nastavit signály MREQ a RD (požadavek na čtení z paměti). Paměť má 1,5 T-stavu na to, aby dopravila byte z požadované adresy na data bus. Během T1 a T2 stavu je také nastaven signál M1, který indikuje fázi načítání instrukce.

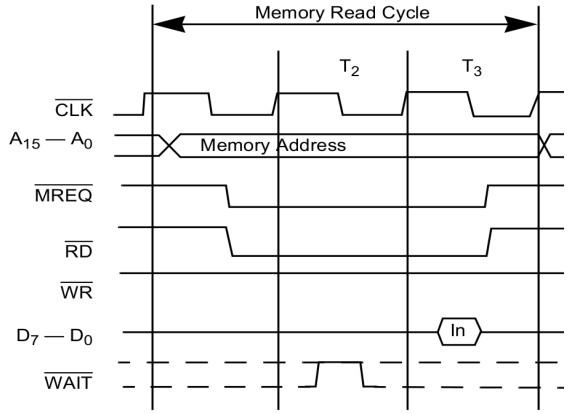
Při náběžné hraně T3 stavu jsou signály M1, MREQ a RD resetovány.

Fáze T3 a T4 stavu je určena k dekódování instrukce. Souběžně s ním je při náběžné hraně T3 stavu na address bus vložena adresa složená z registrů I a R, a tato adresa je použita k aktualizaci paměťového místa nastavením signálů RFSH (při náběžné hraně T3 stavu) a MREQ (při doběžné hraně T3 stavu, po ustálení adresy na address busu).

Signál MREQ je resetován při doběžné hraně T4 stavu, RFSH je resetován při náběžné hraně T-stavu, následujícího po T4 stavu (Pozor: nemusí to být vždy T1 stav – např. instrukce `LD SP,HL` má jediný M-cyklus, který se skládá ze šesti T-stavů!)

Čtení dat z paměti

Načtení dat z paměti je realizováno v rámci minimálně 3 T-stavů:



Obr. 5: Průběh načtení dat z paměti [5]

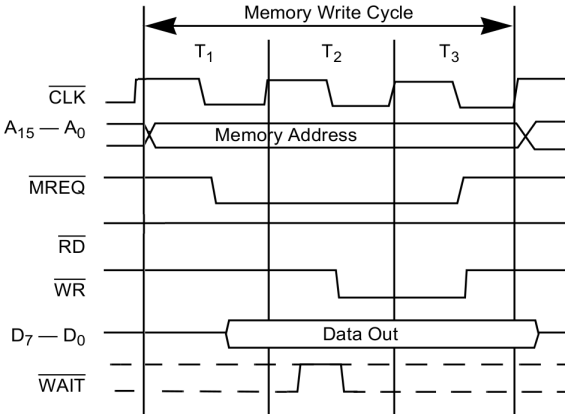
Při náběžné hraně T1 stavu je na address bus umístěna adresa, ze které chceme načíst data. Při doběžné hraně T1 stavu jsou nastaveny signály MREQ a RD, které jsou pamětí chápány jako příkaz ke zkopírování obsahu paměti na adrese uvedené v address busu na data bus.

Data jsou na data busu k dispozici při doběžné hraně T3 stavu, kdy také resetujeme signály MREQ a RD.

V případě, že paměť nestačí dodat data včas, posílá signál WAIT, který je rozpoznáván na začátku každého T-stavu, a který donutí Z80 CPU vykonat jednu instrukci `NOP` navíc. Čas, který má paměť na načtení dat, se tak prodlouží o jeden T-stav.

Zápis dat do paměti

Zápis dat do paměti je realizován v rámci minimálně 3 T-stavů:



Obr. 6: Průběh zápisu dat do paměti [5]

Při náběžné hraně T1 stavu je na address bus umístěna adresa, na kterou budeme zapisovat data. Při doběžné hraně T1 je nastaven signál MREQ a zároveň jsou na data bus zapsána data, která chceme doručit do paměti. Při doběžné hraně T2 stavu jsou data na data busu stabilní a je nastaven signál WR, který je pamětí chápán jako příkaz ke zkopírování obsahu data busu do paměti na adrese uvedené v address busu.

Signály MREQ a WR resetujeme při doběžné hraně T3 stavu.

V případě, že paměť nestačí zapisovat data včas, posílá signál WAIT, který má pro Z80 CPU stejný efekt jako v případě čtení.

1.5.2 Dekódování instrukce

Ve druhé fázi M1 cyklu dochází k dekódování instrukce, která byla načtena z paměti. Ukažme si tento proces na příkladu.

Mějme instrukci assembleru **`LD B,(IX+87)`**.

Tato instrukce má 5 M-cyklů se skladbou T-stavů = {4,4,3,5,3} a je v paměti (řekněme na adrese B3F6h) reprezentována těmito třema byty:

Adresa	Hodnota
B3F6	DD
B3F7	46
B3F8	87

Dekódování a následné provedení instrukce proběhne následovně:

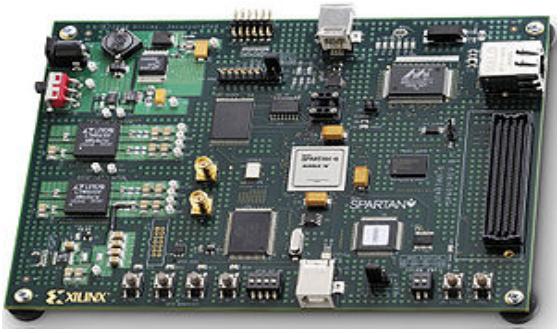
M	T	PC	Akce
1	4	B3F6	Načtení instrukce (DDh)
			PC++
2	4	B3F7	Načtení 2. kódu instrukce (01**000**110b)
			(bity 5-3 určují registr, 000 = regB)
			PC++
3	3	B3F8	Načtení operandu z paměti
			INSTR.d = 87h = -121d
			PC++
4	5	B3F9	Sečtení regIX a INSTR.d
			INSTR.addr = regIX + INSTR.d
5	3	B3F9	regB = INSTR.addr

Příklad je srozumitelný, můžeme tedy přejít k implementaci Z80 CPU na FPGA.

2 Z80 na FPGA

2.1 Co je to FPGA?

Field **P**rogrammable **G**ate **A**rray (FPGA) je programovatelný obvod uzpůsobený k přímé konfiguraci zákazníkem, a to většinou za použití jazyka HDL.
FPGA lze opakovaně libovolně překonfigurovávat a jsou tak (i vzhledem ke své nízké ceně) např. ideálním mezičlánkem mezi návrhem a realizací zákaznických obvodů.



Obr. 7: FPGA [3]

Základem FPGA obvodů jsou [1]

- logické bloky (LE- logic elements resp. LC- logic cells) tvořené logickými prvky (vyhledávacími tabulkami (LUT) a klopnými obvody) spojenými lokálními propojovacími poli
- programovatelná horizontální a vertikální propojení
- programovatelné I/O bloky
- specializované bloky – sčítačky, násobičky, paměti, procesory..

Konfigurace FPGA probíhá sériovým nebo USB kabelem, deska je napájena samostatným zdrojem.

2.2 Co je to VHDL?

Very high speed integrated circuit **H**ardware **D**efinition **L**anguage (VHDL) je programovací jazyk určený k návrhu a testování hardware číslicových systémů. Jazyk byl vytvořen v počátcích 80. let 20. století na objednávku ministerstva obrany USA.

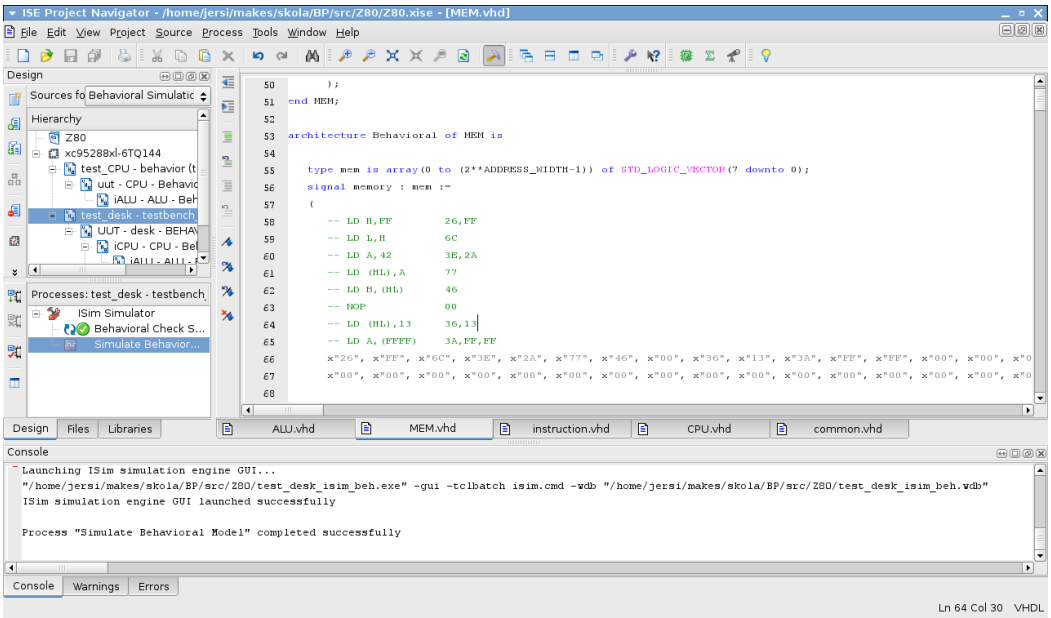
Na rozdíl od běžných programovacích jazyků však výstupem VHDL není spustitelný kód, ale syntetizovaný popis zapojení z hradel a klopných obvodů, určený

pro programovatelné logické obvody nebo jiné typy zákaznických obvodů [1]. Jazyk není závislý na způsobu, jakým bude popisovaný obvod reálně implementován v hardware.

Základním konstrukčním prvkem jazyka VHDL je entita (ENTITY) a její architektura/y (ARCHITECTURE).
Porty entit mohou být vstupní (in), výstupní (out), obousměrné (inout), výstupní se zpětnou vazbou (buffer) nebo neznámé (linkage).
Jednotlivé příkazy jazyka VHDL jsou vykonávány buď paralelně, nebo – v případě použití procesu – sekvenčně.
Samozřejmostí je možnost vytváření procedur a funkcí.
Jazyk VHDL umožňuje vytvářet a linkovat knihovny (LIBRARY) a v nich obsažené knihovní balíky (PACKAGE).

2.3 Vývojové prostředí ISE

Vývojové prostředí ISE nabízí poměrně komfortní způsob vývoje a správy zdrojových VHDL souborů, včetně poměrně robusních nástrojů na testování.



Obr. 8 Ukázka vývojového prostředí ISE

Jak je vidět na obrázku 7, pracovní plocha ISE je rozdělena do tří částí:

- Přehled zdrojů, z nichž je projekt složen
- Okna pro editaci zdrojových VHDL souborů
- Stavová okna pro výpis informací, varování a chyb

Životní cyklus projektu v ISE je potom následující:

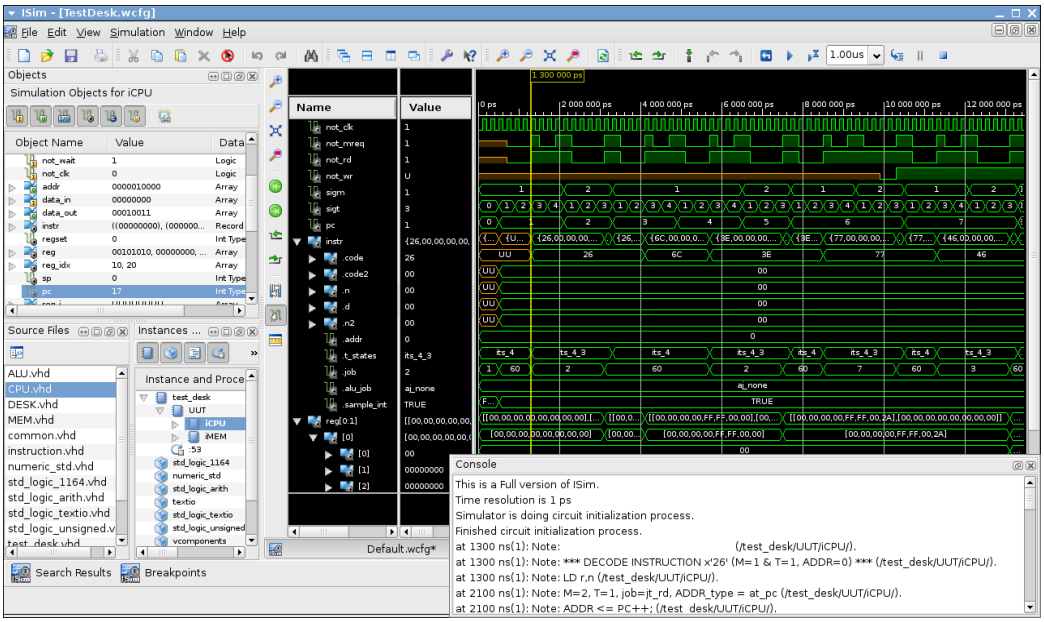
- Analýza požadované funkcionality
- Rozdělení programovaného obvodu na entity a jejich popis, nebo nákres (ISE umožňuje vytvářet a spojovat entity buď přímo popisem ve VHDL, nebo pomocí grafického schema)
- Testování (funkční simulace před syntézou), konfigurování a ladění
- Syntéza
- Případná další funkční simulace po syntéze, popř. časová simulace

V našem projektu se spokojíme s behaviorální simulací před syntézou, a to z několika důvodů:

- Navržený obvod nelze na FPGA Spartan-3 syntetizovat pro jeho rozsáhlost
- I kdybychom obvod syntetizovali, není implementovaná ULA (viz 1.4.3), která by zprostředkovala předávání vstupně-výstupních informací
- Případná časová simulace po syntéze nevypovídá nic o funkční bezchybnosti obvodu

2.4 Testovací prostředí ISim

ISim je velmi výkonný nástroj na testování konfigurovaných obvodů.



Obr. 9: Ukázka výsledku behaviorálního testu v testovacím prostředí ISim

Umožňuje sledovat dění v obvodu a testovat reakce na vnější podněty, uvedené

ve vstupních test bench souborech.

Zvolené signály přehledně zobrazuje na časové ose, jejíž měřítko i rozsah lze podle potřeby měnit.

K doplnění komfortu během simulace je možné v okně Console sledovat debugovací výpisy, zasílané entitami na výstup pomocí příkazu `report`.

3 Behaviorální test modelu

K otestování bezchybné funkčnosti obvodu použijeme krátký testovací program, napsaný v assembleru, na kterém si ukážeme

- způsob implementace jádra Z80 CPU
- vzhled test bench souborů
- sledování navrženého obvodu v prostředí ISim

S odkazem na přiložené zdrojové VHDL kódy si ještě poukažme na klíčová místa implementace:

- model obsahuje tyto vzájemně provázané entity: `Desk`, `CPU`, `ALU` a `MEM`
- součástí knihovny `work` jsou i tyto zdrojové kódy: `common.vhd` a `instruction.vhd`
- bench testy jsou obsaženy v souboru `test_desk.vhd` (všimněte si, že jediným potřebným stimulem je pro nás CLK)

3.1 Testovací program

Následující 12 bytový program v assembleru představuje jednoduchý 2-bitový čítač, realizovaný v registru B:

Adresa	Instrukce	Obsah paměti
0000	LD B,0	06
0001		00
0002	NOP	00
0003	LD A,B	78
0004	INC A	3C
0005	BIT 2,A	CB
0006		57
0007	JR NZ,-9	20
0008		F7
0009	LD B,A	47
000A	JR -9	18
000B		F7

Kód je umístěn v paměti na adresách `0000h` – `000Bh`, demonstrace jeho funkčnosti prokazuje (v souvislosti s přiloženými zdrojovými kódy) zvládnutí

následujících technik:

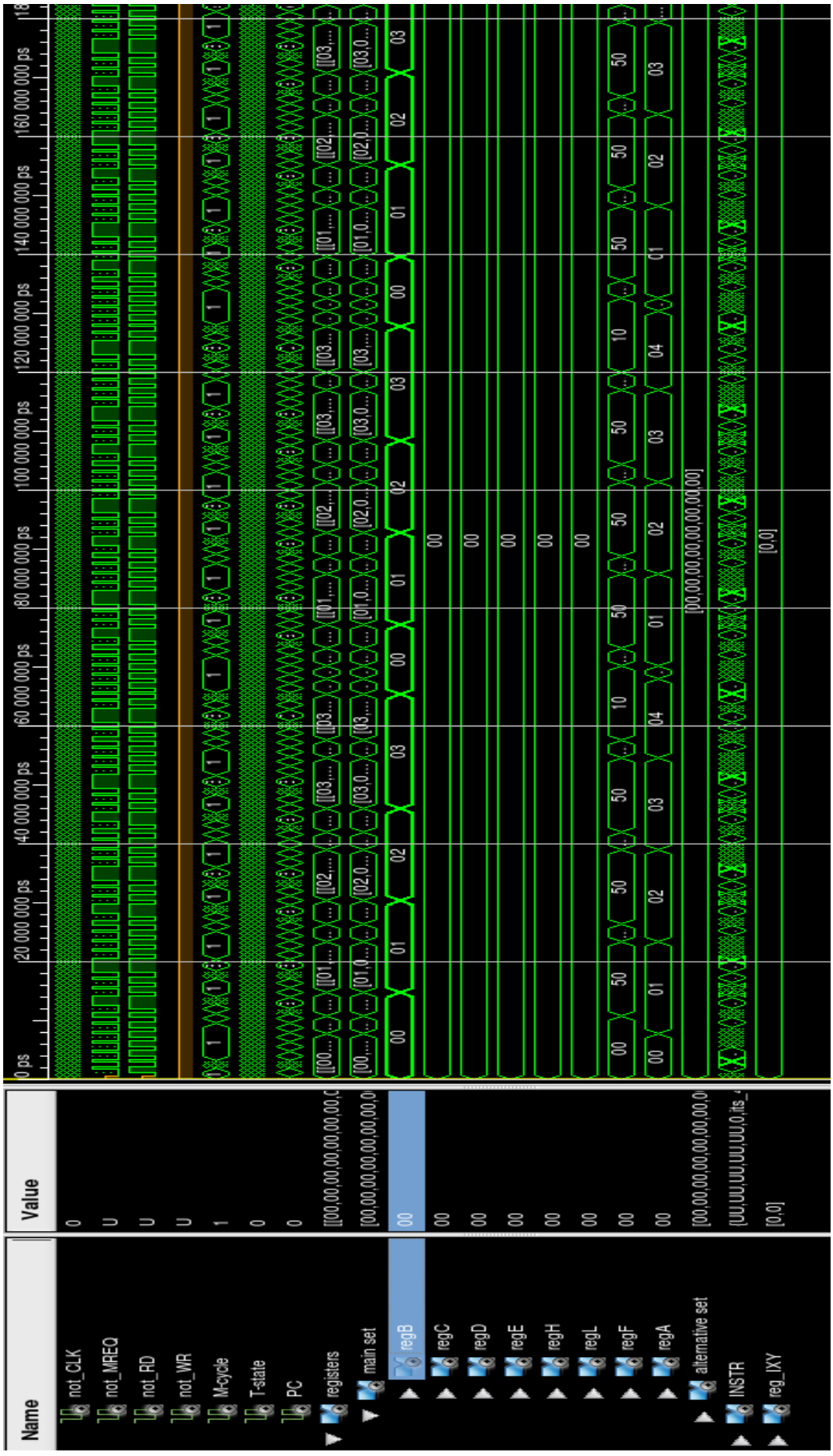
- porozumění chodu procesoru Z80
- správné obsloužení CLK, M-cyklů a T-stavů
- správná implementace registrů, ALU a paměti v jazyce VHDL
- zvládnutí implementace instrukční sady (nebo alespoň její stěžejní části)
- vytvoření test benche a jeho kontrola v ISim

3.2 Výsledky test benche

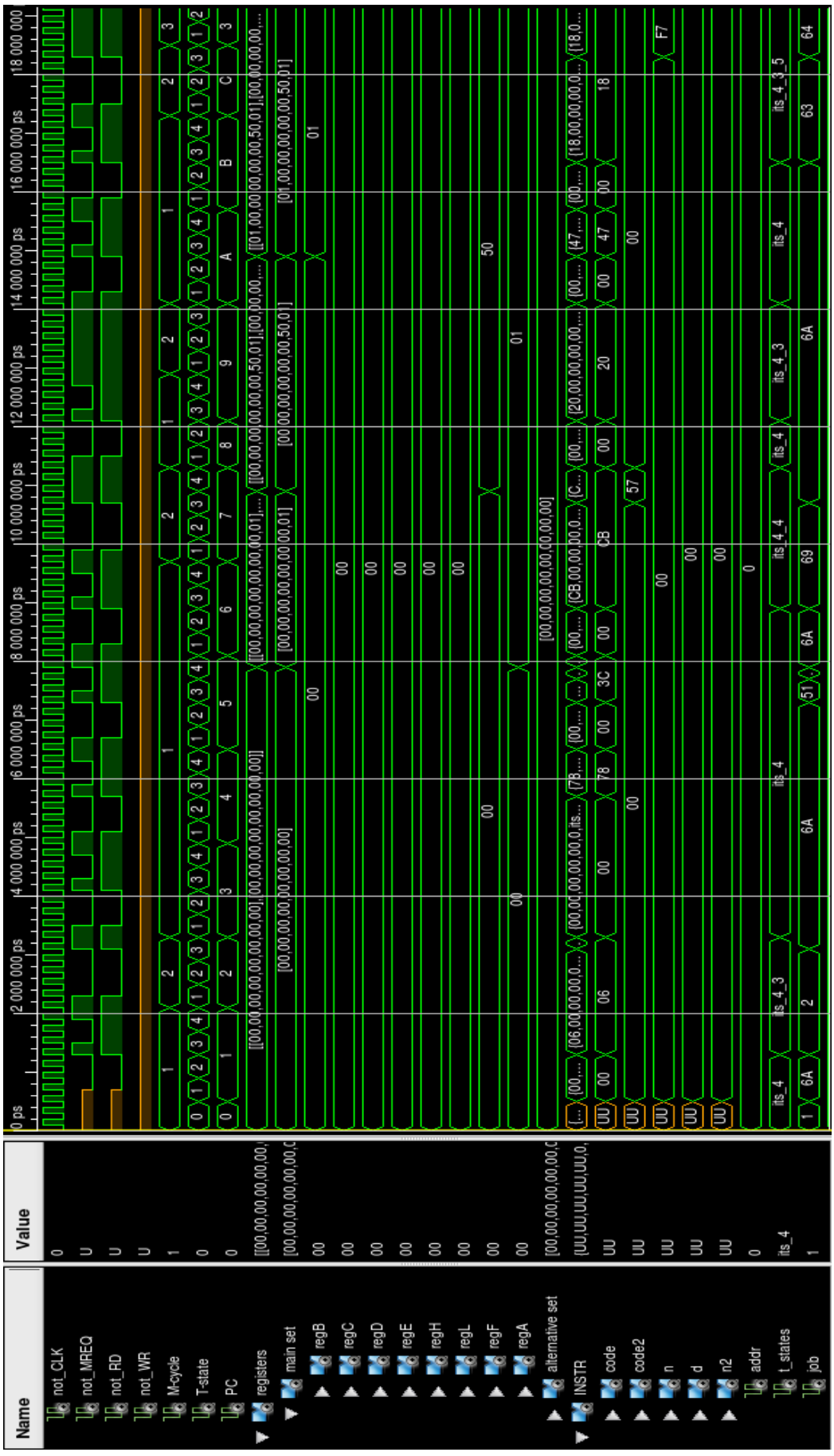
Z obrázků 10 (Pohled na stavové změny v registru B) a 11 (Detailní pohled na průběh programu v prostředí ISim) můžeme vypozorovat, že Z80 CPU zpracovává program následovně:

Instrukce	PC	M	T	Význam
LD B,0	0000	2	(4,3)	Reg[0] <= 0
NOP	0002	1	(4)	No OPeration
LD A,B	0003	1	(4)	Reg[7] <= Reg[0]
INC A	0004	1	(4)	Reg[7] <= Reg[7] + 1
BIT 2,A	0005	2	(4,4)	Pokud 2.bit Reg[7]=0, potom Z <= 1, jinak Z <= 0
JR NZ,-9	0007	12	(4,3,5)	Pokud Z=1, PC <= PC - 9
	nebo	7	(4,3)	
LD B,A	0009	1	(4)	Reg[0] <= Reg[7]
JR -9	000A	12	(4,3,5)	PC <= PC - 9

Další detaily testu viz přiložené zdrojové kódy a kapitola 4.



Obr. 10: Pohled na stavové změny v registru B



Obr. 11: Detailní pohled na průběh programu v prostředí ISim

4 Popis realizace systému

Jako každému, kdo v nějaké oblasti píše svůj první projekt, podařilo se i mně zanést do svého VHDL kódu spoustu nepravostí, rozumějte postupů, které zkušený návrhář hardware nikdy nepoužije. Výsledkem je pak kód, který sice bezvadně splňuje zadání, jeho simulace jsou perfektní, ale jež bychom ani v žertu nenazvali ideálním.

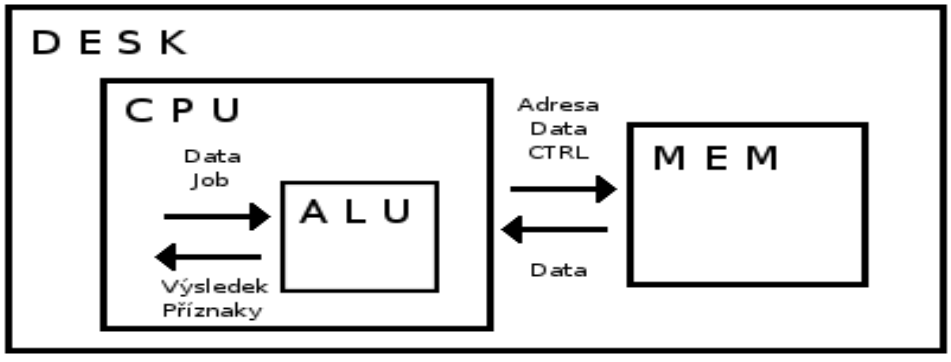
Takový kód je potom často nesyntetizovatelný, a ani ten můj nelze na FPGA Spartan-3 napasovat. Důvodů je několik:

- přílišná volnost v popisu komponent vede k těžko odhadnutelným výsledkům syntézy (např. použití dvojrozměrného pole pro implementaci registrů)
- paměťová komponenta `MEM` není řízena hodinovým signálem `CLK`, ale vstupními signály `NOT_MREQ`, `NOT_RD`, `NOT_WR`, což vede k tvorbě latchů
- malá optimalizace kódu atd..

Smíříme-li se ale s faktem, že dokonalou verzí bude až verze 2., můžeme přistoupit k popisu komponent, ze kterých se mnou implementované jádro Z80 skládá, a k nastínění postupů, jichž jsem použil.

4.1 Obecná charakteristika

Mnou implementované jádro Z80 tvoří ***4 komponenty***, zobrazené na obrázku:



Obr. 12: Blokové schema jádra Z80

Komponenta `DESK` zapouzdřuje komponenty `CPU` a `MEM`, zajišťujíc komunikaci mezi nimi. Komponenta `CPU` zapouzdřuje komponentu `ALU` a komunikuje s komponentou `MEM`.

Kromě zdrojových kódů těchto 4 komponent sestává projekt z dalších 3 souborů: `common.vhd`, `instruction.vhd` a `test_desk.vhd`.

Rozeberme si nyní všechny tyto součásti podrobněji:

Komponenta **DESK**

- zapouzdřuje komponenty `CPU` a `MEM` a zprostředkovává mezi nimi komunikaci
- obsahuje vstupní hodinový signál `CLK`, který předává komponentě `CPU`

Komponenta **CPU**

- jádro Z80 CPU
  - realizuje načtení, dekódování a provedení instrukce (implementováno celkem 22% instrukční sady)
- implementuje
  - programový čítač (**PC**), adresu vrcholu haldy (**SP**), Memory refresh (**R**) a Interrupt vector (**I**)
  - registry - implementovány jako 2-rozměrné pole **REG[***RegSet*, *Registr***]**, kde
    - *RegSet* (0,1) slouží k přepínání mezi hlavní a alternativní sadou registrů během přerušení
    - *Registr* (B=0, C=1, D=2, E=3, H=4, L=5, F=6, A=7) je offset bytu (`STD_LOGIC_VECTOR(7 downto 0)`), reprezentujícího konkrétní registr

    Např. registr B v hlavní sadě je reprezentován signálem REG[0,0], registr H v alternativní sadě je reprezentován signálem REG[1,4] atd.
  - signál **INSTR**, který nese informaci o zpracovávané instrukci:

`code, code2`	1. (případně i 2. kód) instrukce
`n, d, n2`	instrukční operandy
`addr`	interní adresování instrukce
`t_states`	počty T-stavů jednotlivých M-cyklů instrukce
`job`	ukazatel do seznamu instr. příkazů JOBS
`alu_job`	řídící signál pro ALU
`sample_int`	povolení samplování přerušení
- zapouzdřuje komponentu `ALU`, se kterou komunikuje pomocí 2 datových,

1 řídícího a 1 flag signálu; výstupem z `ALU` je 1 datový a 1 flag signál

- komunikuje s pamětí (`MEM`) pomocí 4 řídících, 1 adresního a 1 datového signálu; výstupem z paměti je 1 řídící a 1 datový signál

Komponenta **ALU**

- realizuje (početní) operace mezi dvěma dodanými signály.
  Implementovány jsou tyto operace:
  - ADD
    sečte dva vstupní signály a podle výsledku nastaví flags
  - INC
    ke 2. vstupnímu signálu přičte jedničku a podle výsledku nastaví flags
  - další implementované operace jsou interní (LD_AIR, ROUTE apod.)

Komponenta **MEM**

- čte/zapisuje data z/do paměti na základě 3 negovaných řídících signálů `NOT_MREQ`, `NOT_RD` a `NOT_WR`, 2 datových (I/O) a 1 adresního signálu
- výstupní řídící signál `NOT_WAIT` není obsloužen

Soubor **common.vhd**

- obsahuje sadu obecně použitých konstant a funkcí

Soubor **instruction.vhd**

- obsahuje konstanty pro zpracování instrukcí
- implementuje konstantní pole JOBS, které definuje způsob zpracování jednotlivých instrukčních M-cyklů

Soubor **test_desk.vhd**

- test bench – buzení `CLK` signálu pro testování komponennty `DESK` v ISim.

Pozn: pro zjištění přesných charakteristik signálů a pro ucelený náhled na kód vizte zdrojové soubory.

4.2 Rozbor instrukce LD B, (IX + 87)

Dříve než přistoupíme k detailnímu rozboru testovacího programu, projděmež zahřívacím kolem v podobě detailního rozboru instrukce **`LD B,(IX+87)`**, zmiňované již jednou v kap.1.5.2 na str. 20.

Tato jednoduchá instrukce má 5 M-cyklů se skladbou T-stavů = {4,4,3,5,3} a je kódovaná těmito 3 byty: `DD`, `46` a `87`.

Jejich význam je následující:

hex	bin	význam
DD	11011101	DD => práce s 16-bit registrem IX
46	01**000**110	01**rrr**110 => LD instrukce, kde obsah IX je bází adresy a cílem je registr, jehož offset je složen z bitů 5-3 (000 = registr B)
87	10000111	signed displacement od báze adresy (IX), tj. IX + 87h neboli IX - 121d

Abychom si tuto instrukci předvedli nad nějakou smysluplnou adresou (displacement `87h` by nás z důvodu podtečení odkázal někam do horní části paměti), předsuneme jako první instrukci **`LD IX,7A`**, čímž zajistíme, že výsledná adresa `IX+87 = 0001h`.

Instrukce umístíme od adresy `0000h` (na této adrese normálně začíná ROM, ale my si to můžeme dovolit), takže obsah paměti bude vypadat takto:

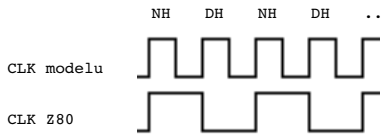
adresa	obsah	instrukce
0000	DD	LD IX,007Ah
0001	21	
0002	7A	
0003	00	
0004	DD	LD B,(IX+87)
0005	46	
0006	87	

Z výše uvedeného vyplývá, že výsledným obsahem registru B bude obsah paměti na adrese `0001h`, což je číslo `21h`.

Významnými momenty, kterých bychom si měli na tomto příkladu všimnout, jsou zejména:

- `NOT_CLK`

  Signál `CLK` (resp. v modelu použitý negovaný `NOT_CLK`) je v modelu "zdvojený" (na 1 T-stav připadnou 2 tiknutí) – je to způsobeno tím, že ve VHDL je lepší držet se jen náběžné hrany hodinového signálu. Abychom ale byli schopni nastavovat signály i na doběžné hraně, nechal jsem tikat hodiny modelu s dvojnásobnou frekvencí: první náběžná hrana `CLK` odpovídá náběžné hraně (NH) CLK Z80, druhá pak doběžné hraně (DH) CLK Z80 [5]. Na funkčnost modelu však toto řešení nemá žádný vliv.



Obr. 13: Dvojnásobná frekvence CLK modelu oproti CLK Z80

- fáze načtení instrukce a operandu

  Všimněte si, že průběh nastavování signálů `MREQ` a `RD` je shodný s průběhem, popsaným v kap. 1.5.1 na str. 18.

- rozdělení instrukcí do M-cyklů a T-stavů

  Striktně jsem dodržoval počty M-cyklů a jejich T-stavů podle dokumentace Z80 CPU [5].

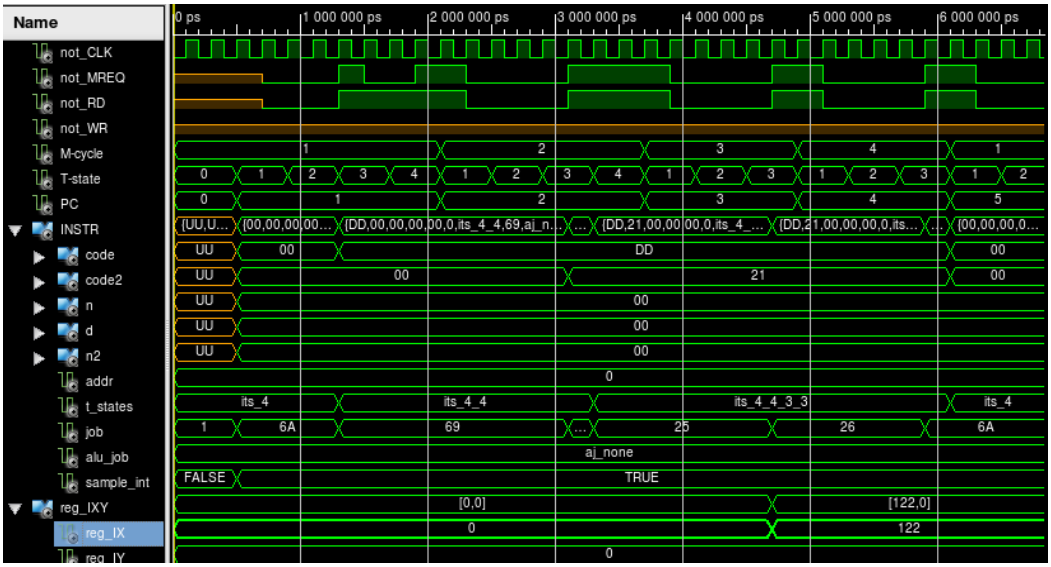
- 16-bitové operandy jsou v paměti (narozdíl od big-endian notace v případě registrů) uloženy v notaci little-endian, tzn. nižší polovina čísla je v nižším byte paměti. Číslo `007Ah` je tedy na adrese `0002` umístěno takto:

adresa	obsah
0002	7A
0003	00

Přejděme tedy k popisu jednotlivých fází na ukázkách z výstupu testovacího prostředí ISim.

Instrukce `LD IX,7A`

Průběh zpracování instrukce `LD IX,7A` v ISim je zobrazen na obrázku:



Obr. 14: Průběh instrukce LD IX,7A

M	T	PC	Akce
1	4	0000	Načtení instrukce (DDh)
			PC++

Během M1 cyklu dochází k načtení instrukce z paměti `MEM` do `INSTR.code` (NH T1 – DH T2) a k jejímu dekódování (NH T3).

Během T1 dále dochází při NH k inkrementaci `PC` o 1 a při DH k resetu signálu `NOT_M1` (zpět na hodnotu 1 je nastaven při DH T2).

Načtený opcode DD předznamenává rodinu instrukcí pracujících s 16-bit registrem IX (je jich celkem 19), konkrétní instrukce je identifikována na základě 2. načteného opcode během M2.

M	T	PC	Akce
2	4	0001	Načtení 2. kódu instrukce (21h)
			PC++

Během M2 cyklu dochází k načtení 2. kódu instrukce z paměti `MEM` do `INSTR.code2` (NH T1 – DH T2) a k jejímu dekódování (NH T4), kdy jsou nastaveny tyto hodnoty:

- `INSTR.t_states=its_4_4_3_3`

  Pole `t_states` udává, kolik T-stavů má každý z M-cyklů zpracovávané instrukce. Konstanta `its_4_4_3_3` říká, že M1 má 4 T-stavy, M2 4, M3 3 a

M4 má 3 T-stavy. Tyto počty pochází z dokumentace Z80 CPU, ale dají se snadno odvodit – 4 T-stavy jsou použity pro načtení a dekódování instrukce, při prostém načítání operandu z paměti vystačíme se 3 T-stavy.

- `INSTR.JOB=c_LD_IXY_nn`

  `JOB` je ukazatel do pole mikroinstrukcí, které mají následující strukturu:

`M-cycle`	M-cyklus, během kterého bude JOB proveden
`T-state`	T-stav M-cyklu, během kterého bude JOB proveden
`job type`	Typ JOBu (čtení/zápis z/do `MEM`/`ALU`, `PC` adjustment nebo sestavení adresy jako báze + displacement)
`addr.type`	typ adresního ukazatele
`addr.pointer`	adresní ukazatel
`data type`	typ datového ukazatele
`data pointer`	datový ukazatel
`lo–hi`	použít nižší/vyšší byte (v případě 16-bit operací)
`alu job`	typ operace při práci s `ALU`

Během NH T1 dále dochází k inkrementaci `PC` o 1.

M	T	PC	Akce
3	3	0002	Načtení nižší části operandu (7Ah)
			PC++

Během M3 cyklu dochází ke zpracování 1. části JOBu `c_LD_IXY_nn` (který realizuje načtení 16-bitového operandu z paměti do registru IX / IY):

```
(3, 1, jt_RD, at_PC, "110", at_INSTRcode2_RegPair, rps_fromCode, regPair_LO, aj_NONE)
```

Výklad: Během M3,T1 načti z `MEM` (adresa uvedena v `PC`, jeho hodnotu po načtení zvyš o 1) byte a ulož ho do nižší části registru, který dohledáš takto:

- bity 5–4 `INSTR.code2` udávají offset 16-bit registru v *sadě registrů*, kterou složíme z adresního a datového ukazatele (v tomto případě z bitů 6–5 `INSTR.code`)
- možnými *sadami registrů* jsou:

  ```
  00 :  BC, DE, HL, SP
  01 :  BC, DE, HL, AF
  10 :  BC, DE, IX, SP
  11 :  BC, DE, IY, SP
  ```

Výsledkem je v tomto konkrétním případě zápis načteného operandu do nižší části registru IX.

Samotné načtení operandu je implementováno v souladu s časováním uvedeným v kap. 1.5.1 na str. 19.

M	T	PC	Akce
4	3	0003	Načtení vyšší části operandu (00h) PC++

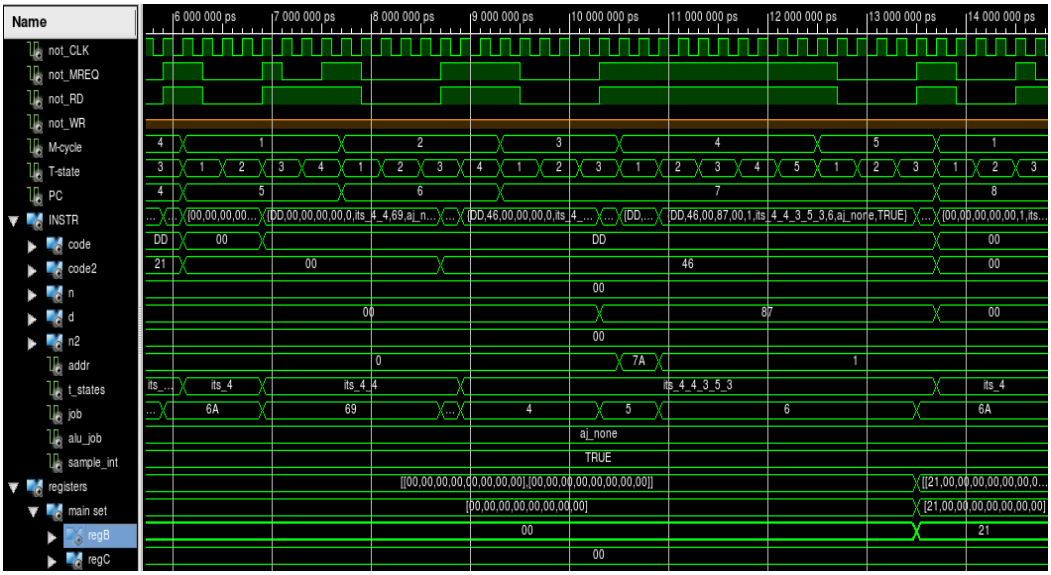
Během M4 cyklu dochází ke zpracování 2. části JOBu `c_LD_IXY_nn`:

```
(4, 1, jt_RD, at_PC, "110", at_INSTRcode2_RegPair, rps_fromCode, regPair_HI, aj_NONE)
```

Načtení operandu do horní části registru IX je implementováno v souladu s časováním uvedeným v kap. 1.5.1 na str. 19.

Instrukce `LD B,(IX+87)`

Průběh zpracování instrukce `LD B,(IX+87)` v ISim je zobrazen na obrázku:



Obr. 15: Průběh instrukce LD B,(IX+87)

Jako každá instrukce začíná i tato M1 stavem, během kterého je do `INSTR.code` načten opcode a dochází k jeho dekódování:

M	T	PC	Akce
1	4	0004	Načtení instrukce (DDh) PC++

Opcode `DD` předznamenává práci s registrem IX. Přesnou identifikaci instrukce

provedeme až na základě 2. opcode, který načteme do `INSTR.code2` během M2:

M	T	PC	Akce
2	4	0005	Načtení 2. kódu instrukce (01**000**110b) (bity 5-3 určují registr, 000 = regB) PC++

Po identifikaci `LD r,(IX+d)` instrukce nastavíme během M2 T4 počty T-stavů jako `INSTR.t_states=its_4_4_3_5_3` a do `INSTR.job` vložíme ukazatel na první JOB (konstanta `c_LD_r_pIXYd`).

M	T	PC	Akce
3	3	0006	Načtení operandu z paměti PC++ INSTR.d = 87h = -121d

Během M3 provádíme tento JOB:

```
(3, 1, jt_RD, at_PC, "000", at_INSTR_d, "000", regPair_NONE, aj_NONE)
```

Význam: do `INSTR.d` vlož obsah paměti z adresy, uvedené v `PC`; `PC` zvyš o 1 (`INSTR.d <= MEM[PC++]`).

Pozn: Pokud bychom chtěli směr zápisu otočit, museli bychom namísto JOB type `jt_RD` použít `jt_WR`. Pak by došlo k zápisu `MEM[PC++] <= INSTR.d`.

M	T	PC	Akce
4	5	0007	INSTR.addr = regIX + INSTR.d

Během M4 provádíme tento JOB:

```
(4, 1, jt_aXYd, at_NULL, "000", at_NULL, "000", regPair_NONE, aj_NONE)
```

Význam: do `INSTR.addr` vlož součet `IX + d` (`INSTR.addr <= IX|IY+d`).

M	T	PC	Akce
5	3	0007	regB = INSTR.addr

Během M5 provádíme tento JOB:

```
(5, 1, jt_RD, at_INSTR_addr, "000", at_INSTRcode2_Reg, "101", regPair_NONE, aj_NONE)
```

Význam: `REG[INSTR.code2("101" downto "101"-2)] <= INSTR.addr`

Rozbor: `REG[INSTR.code2("101" downto "101"-2)] =`

```
REG[INSTR.code2(5 downto 3)] =
REG["01000110"(5 downto 3)] =
REG["000"] = registr B
```

Výsledkem tohoto JOBu je tedy operace `regB <= INSTR.addr`.

Jak je možno vidět na *Obr. 15*, obsahem registru B po doběhnutí 2. instrukce je obsah paměti na adrese `0001`, tedy číslo `21h`.

4.3 Detailní rozbor testovacího programu

Na základě výše uvedeného popisu implementace Z80 CPU si nyní detailně rozebereme testovací program, zmíněný v kap. 3.1 na str. 25.

Tento program je uložen v paměti od adresy `0000h` a jeho výpis je následující:

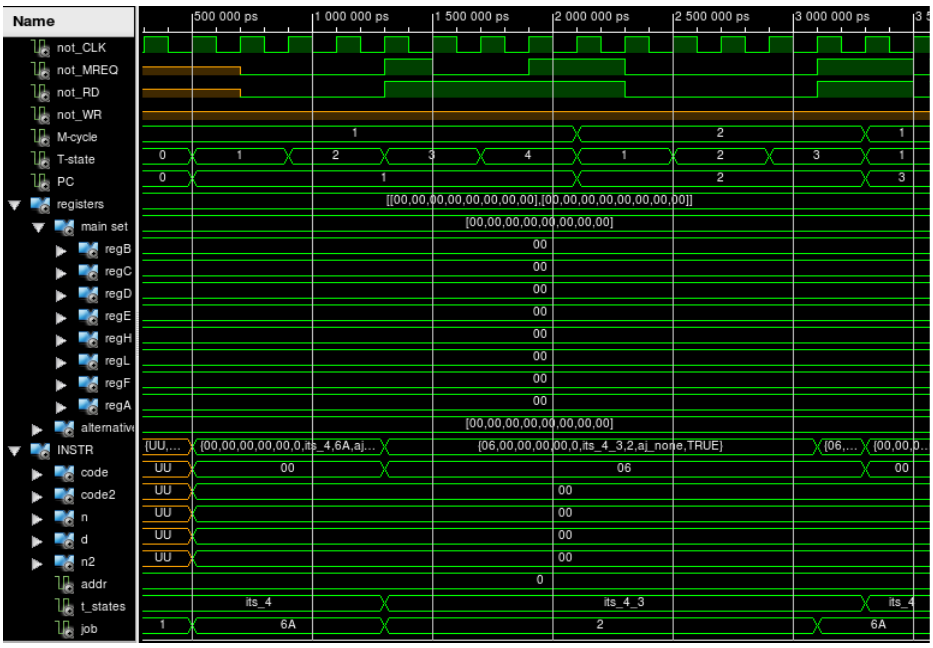
Adresa	Obsah	Instrukce
0000	06	LD B,0
0001	00	
0002	00	NOP
0003	78	LD A,B
0004	3C	INC A
0005	CB	BIT 2,A
0006	57	
0007	20	JR NZ,-9
0008	F7	
0009	47	LD B,A
000A	18	JR -9
000B	F7	

Abychom stále neopakovali již jednou vysvětlené, předesílám, že instrukce jsou řešeny – tak jak jsme si předvedli v předchozí kapitole – formou JOBS mikro-instrukcí.

Správnost řešení své bakalářské práce dokládám přiloženými ukázkami z ISim a průvodními poznámkami o použitých JOBech. Instrukce jsou dále doplněny o vysvětlení, jakou konkrétní roli hrají v testovacím programu.
Pro detailní pochopení implementace nahlédněte, prosím, do přiložených zdrojových kódů.

`LD B,0` `(RegB <= 00h)`

Instrukce nahraje do registru B číslo `00h`. Jedná se o reset registru B, který je volán na začátku programu a pokaždé, když hodnota registru A překročí hodnotu 3.



Obr. 16: Průběh instrukce LD B,0

M	T	PC	Akce
1	4	0000	Načtení instrukce (06h)
			PC++

M	T	PC	Akce
2	3	0001	Načtení operandu z paměti (00h)
			PC++
			regB <= 00h

K načtení operandu z adresy do registru je během M2 použit tento JOB:

```
(2, 1, jt_RD, at_PC, "000", at_INSTRcode_Reg, "101", regPair_NONE, aj_NONE)
```

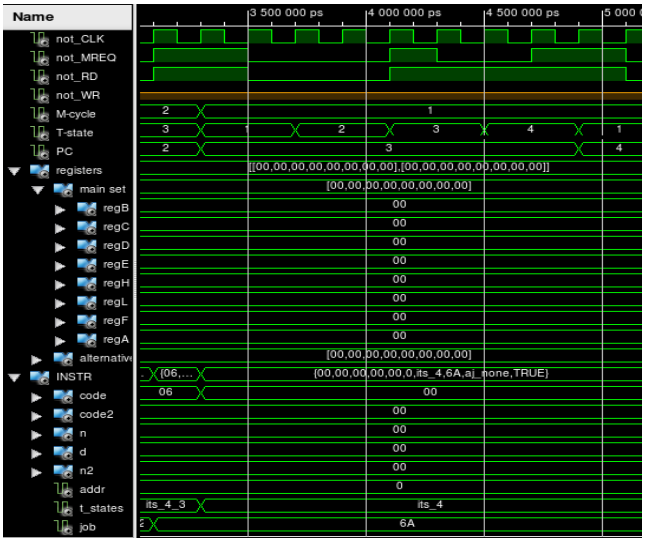
Cílový registr je zakódován v opcode (`INSTR.code`) takto:

`06h = 00`**`000`**`110b = 00`**`rrr`**`110`

Registrem `Reg[0]` je regB.

NOP (No OPeration)

Ukázka operace NOP, která nedělá nic.

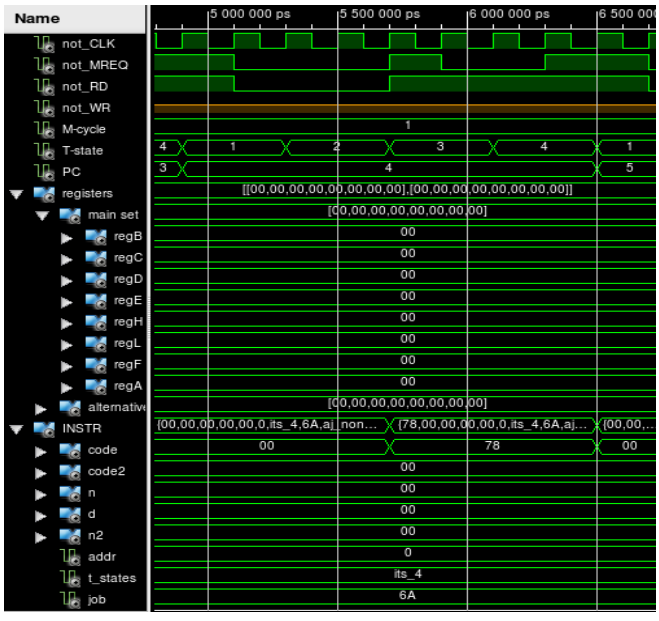


Obr. 17: Průběh instrukce NOP

M	T	PC	Akce
1	4	0002	Načtení instrukce (00h)
			PC++

LD A,B (RegA <= RegB)

Instrukce nahraje do registru A obsah registru B. Reg A používáme k inkrementování hodnoty, která je uložena v reg B. Pokud tato hodnota přeteče ze 3 na 4 (test 2. bitu reg A nastaví Z-flag <= 0), nastavíme `PC <= 0` (reset reg B).



Obr. 18: Průběh instrukce LD A, B

M	T	PC	Akce
1	4	0003	Načtení instrukce (78h)
			PC++
			RegA <= RegB

Zatím nezoptimalizovaná instrukce, nepoužívá žádný JOB.

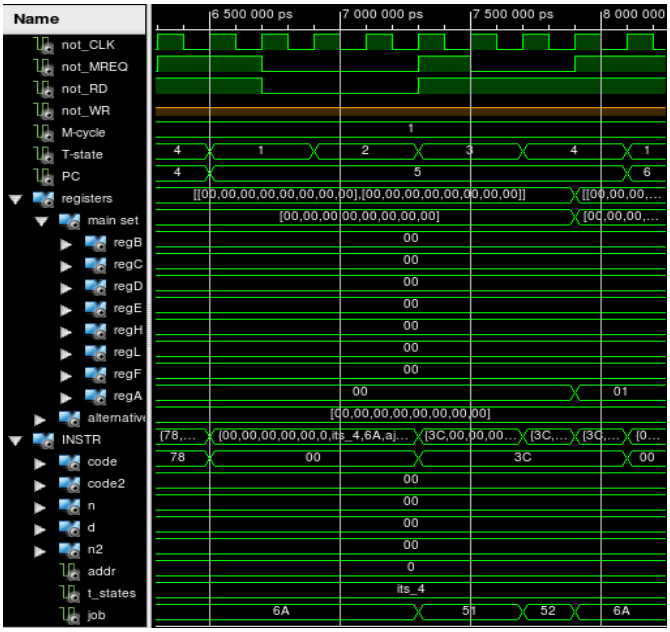
Zdrojový a cílový registr jsou zakódovány v opcode (`INSTR.code`) takto:

78h = 01**111000**b = 00**dddsss**

kde `ddd` je offset cílového registru a `sss` je offset zdroje.

`INC A` `(RegA++)`

Instrukce zvýší hodnotu registru A o 1.



Obr. 19: Průběh instrukce INC A

M	T	PC	Akce
1	4	0004	Načtení instrukce (3Ch)
			PC++
			RegA++

Instrukce využívá k inkrementování obsahu registru ALU, jíž pošle jako parametr číslo registru, obsažené v opcode (`3Ch = 00111100b = 00rrr100`).

Komunikace s ALU je implementována pomocí těchto JOBů během T4 M1:

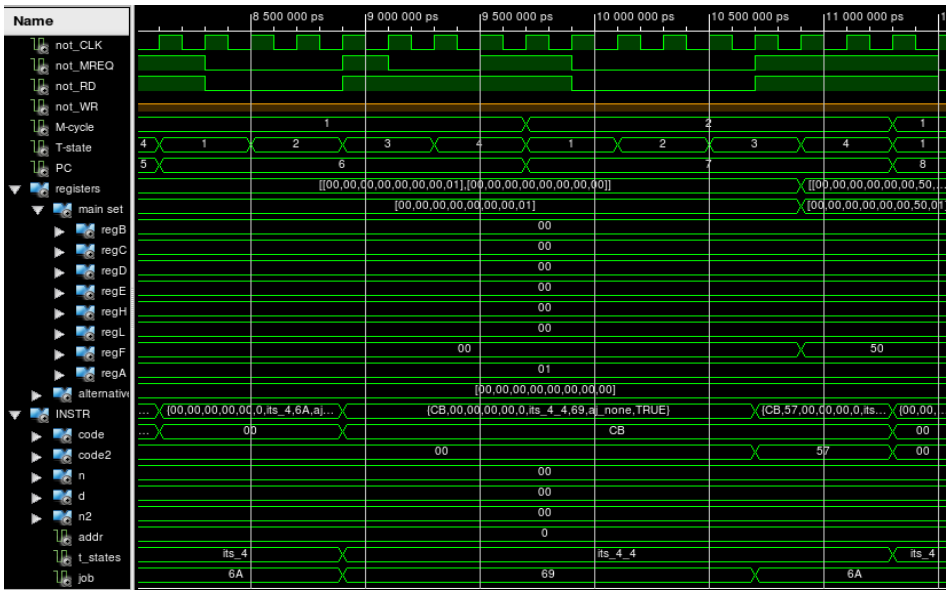
```
(1, 4, jt_ALU_WR, at_NULL, "000", at_INSTRcode_Reg, "101", regPair_NONE, aj_INC)
(1, 4, jt_ALU_RD, at_NULL, "000", at_INSTRcode_Reg, "101", regPair_NONE, aj_NONE)
```

BIT 2,A (Test 2.bitu RegA s nastavením Z-flag)

Testujeme 2.bit registru A (pozn: nejméně významný je bit 0).

Pokud je tento bit = 0, pak nastavíme Z-flag <= 1, jinak nastavíme Z-flag <= 0

(je-li hodnota 2.bitu rovna 1, došlo k přetečení hodnoty regA ze 3 na 4).



Obr. 20: Průběh instrukce BIT 2,A

M	T	PC	Akce
1	4	0005	Načtení instrukce (CBh)
			PC++

M	T	PC	Akce
2	4	0006	Načtení 2. kódu instrukce (57h)
			PC++

Zatím nezoptimalizovaná instrukce, nepoužívá žádný JOB.

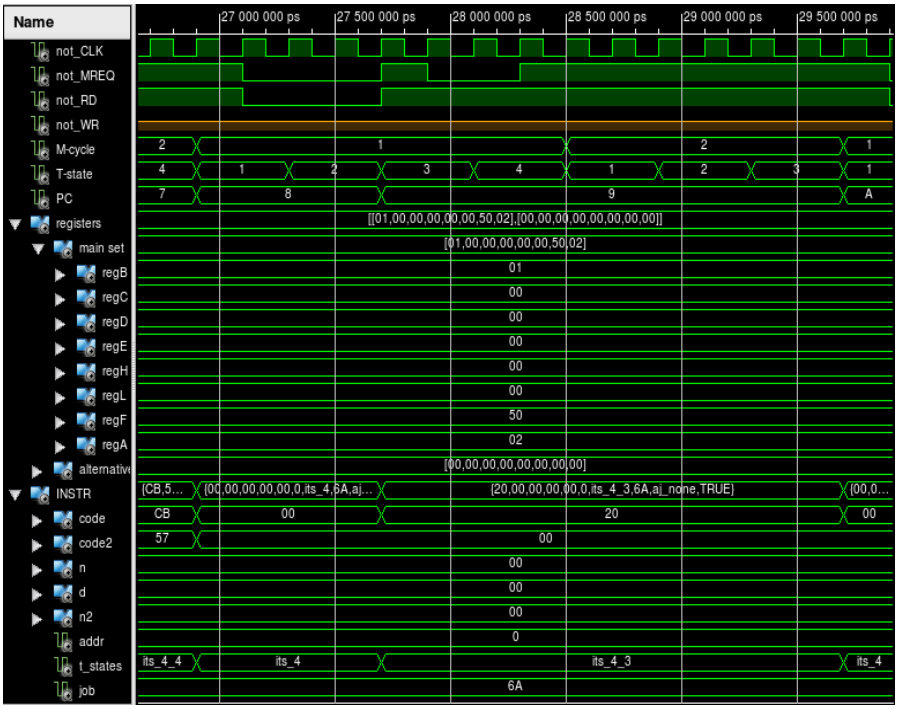
Bit a registr jsou zakódovány ve 2. opcode (`INSTR.code2`) takto:

57h = 01**010111**b = 01**bbbrrr**

kde `bbb` je číslo bitu a `rrr` offset registru.

JR NZ,-9 (Pokud Z-flag=1)

Pokud je Z-flag = 1, pokračujeme ve vykonávání programu další instrukcí.



Obr. 21: Průběh instrukce JR NZ, -9 (pokud Z-flag = 1)

Instrukce `JR NZ,e` má dvě různé podoby M-cyklů a jejich T-stavů:

T={4,3} pokud je Z-flag = 1

T={4,3,5} pokud je Z-flag = 0

Společnou částí je načtení opcode a operandu:

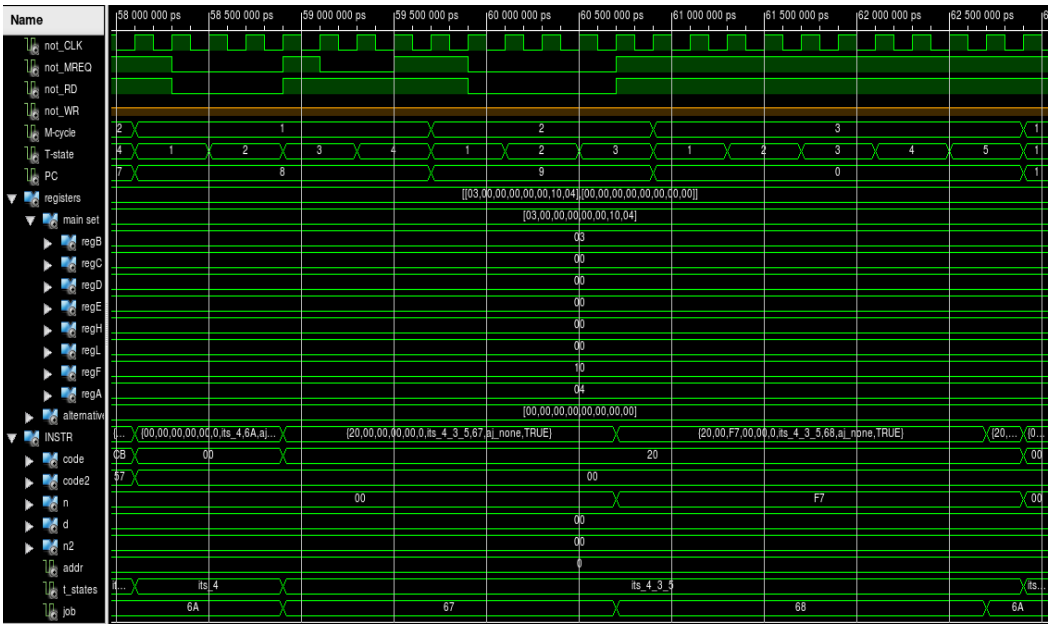
M	T	PC	Akce
1	4	0007	Načtení instrukce (20h)
			PC++

M	T	PC	Akce
2	3	0008	Načtení operandu z paměti (F7h)
			PC++

Po načtení operandu dojde k vyhodnocení podmínky Z-flag = 1, v tomto případě se operand zahazuje a program pokračuje dál na adrese uvedené v `PC`.

`JR NZ,-9 (Pokud Z-flag=0)`

Pokud je Z-flag = 0 (došlo k přetečení hodnoty regA ze 3 na 4), snížíme hodnotu `PC` o 9, čímž dojde ke skoku na adresu `0000h` a k resetu regB.



Obr. 22: Průběh instrukce JR NZ, -9 (pokud Z-flag = 0)

M	T	PC	Akce
1	4	0007	Načtení instrukce (20h)
			PC++

M	T	PC	Akce
2	3	0008	Načtení operandu z paměti (F7h)
			PC++
			INSTR.n <= F7h (-9d)

Načtení operandu a jeho uložení do `INSTR.n` je během M2 implementováno pomocí tohoto JOBu:

```
(2, 1, jt_RD, at_PC, "000", at_INSTR_n, "000", regPair_NONE, aj_NONE)
```

M	T	PC	Akce
3	5	0009	PC <= PC + INSTR.n (-9d)

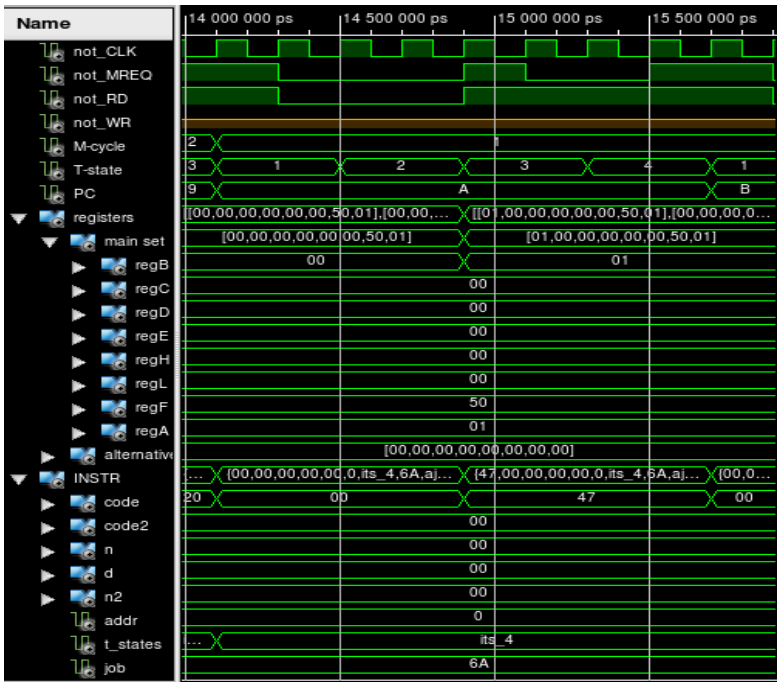
Změna hodnoty `PC` během M3 je implementována pomocí tohoto JOBu:

```
(3, 1, jt_adjPC, at_NULL, "000", at_NULL, "000", regPair_NONE, aj_NONE)
```

Pozn: pochopitelně že by bylo možné změnit `PC` v kratším čase; 5 T-stavů je ovšem doba uvedená v dokumentaci Z80 CPU [5]..

`LD B,A` `(RegB <= RegA)`

Pokud nedošlo k přetečení regA na hodnotu 4, zapíšeme inkrementovanou hodnotu regA zpět do regB.



Obr. 23: Průběh instrukce LD B, A

M	T	PC	Akce
1	4	0009	Načtení instrukce (47h)
			PC++
			RegB <= RegA

Zatím nezoptimalizovaná instrukce, nepoužívá žádný JOB.

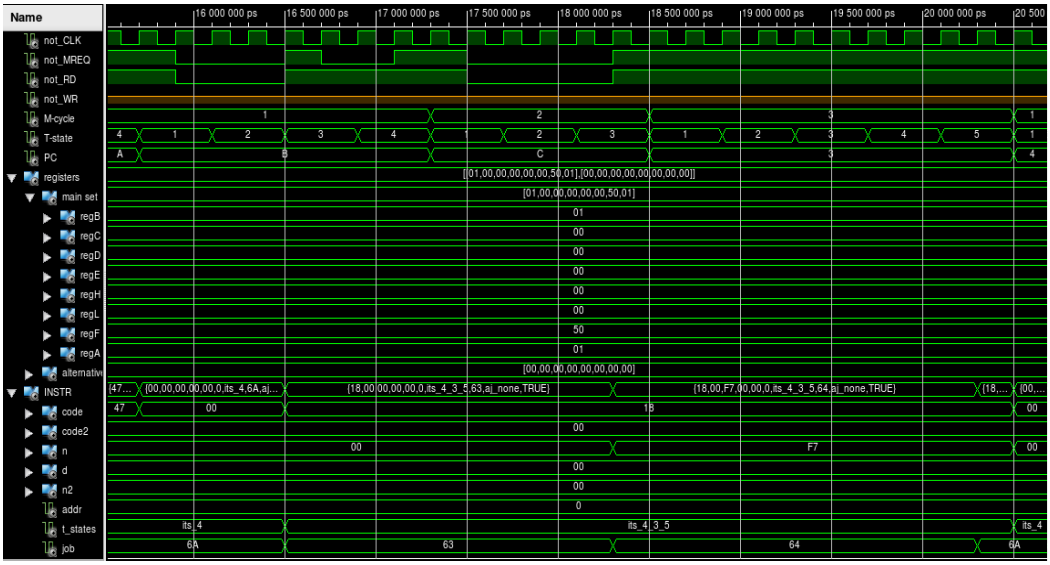
Zdrojový a cílový registr jsou zakódovány v opcode (`INSTR.code`) takto:

47h = 01**000111**b = 00**dddsss**

kde `ddd` je offset cílového registru a `sss` je offset zdroje.

JR -9 (PC <= PC - 9)

Nepodmíněný relativní skok o 9 bytů vzad, tj. na adresu `0003h` (což je místo hned za resetem regB a instrukcí NOP).



Obr. 24: Průběh instrukce JR -9

M	T	PC	Akce
1	4	000A	Načtení instrukce (18h)
			PC++

M	T	PC	Akce
2	3	000B	Načtení operandu z paměti (F7h = -9d)
			PC++
			PC <= PC - 9

Načtení operandu a jeho uložení do `INSTR.n` je během M2 implementováno pomocí tohoto JOBu:

```
(2, 1, jt_RD, at_PC, "000", at_INSTR_n, "000", regPair_NONE, aj_NONE)
```

M	T	PC	Akce
3	5	0009	PC <= PC + INSTR.n (-9d)

Změna hodnoty `PC` během M3 je implementována pomocí tohoto JOBu:

```
(3, 1, jt_adjPC, at_NULL, "000", at_NULL, "000", regPair_NONE, aj_NONE)
```

Pozn: pochopitelně že by bylo možné změnit `PC` v kratším čase; 5 T-stavů je ovšem doba uvedená v dokumentaci Z80 CPU [5]..

Závěr

Nastudoval jsem rodinu 8-bitových procesorů firmy ZiLOG se zaměřením na slavný Z80 CPU, který popisuji v kapitole 1.
Nastudoval jsem jazyk VHDL a napsal v něm jádro Z80 CPU.
V jazyce symbolických adres jsem sestavil jednoduchý program, který realizuje 2-bitový čítač v registru B, a tento příklad jsem detailně rozebral v kapitole 4.2.
Návrh jsem odsimuloval v prostředí ISim.
Z důvodu užití nevhodných konstrukcí a postupů v jazyce VHDL je však výsledný syntetizovaný kód nerealizovatelný na FPGA Spartan-3 pro svoji rozsáhlost. Další optimalizací kódu VHDL by jistě bylo možné výsledek syntézy významnou měrou ovlivnit.

Tato bakalářská práce mi umožnila detailneji nahlédnout do tajů mikroprocesoru Z80 firmy ZiLOG a naučit se, jakým způsobem probíhá návrh integrovaných obvodů jazykem VHDL za použití nástrojů ISE a ISim firmy Xilinx.

Nad prací jsem strávil necelý půlrok svého života, ale ani tak se mi nepodařilo dotáhnout projekt do té podoby, kdy by k FPGA desce byl připojen monitor a klávesnice, abych ctěné komisi mohl předvést hraní her typu Saboteur či Manic Miner (jimž jsem obětoval kus svého mládí).

V žádném případě však docílený stav nepovažuji za neúspěch – naopak!
Naučil jsem se spoustu nových věcí (principy číslicové techniky, VHDL), procvičil se v programování a v neposlední řadě si i občerstvil a dále prohloubil své znalosti Z80 CPU a assembleru.

Tato práce může být chápána i jako základ pro práci někoho dalšího, kdo bude mít čas a trpělivost dopracovat do modelu překlad zbytku instrukční sady, zpracování přerušení, pro odvážné se nabízí i možnost implementace zákaznického obvodu ULA..

Literatura

[1] Pinker Jiří, Poupa Martin: Číslicové systémy a jazyk VHDL, BEN 2006, ISBN 80-7300-198-5

[2] Rozkovec Martin: Návrh softwarového procesoru v FPGA obvodu, diplomová práce TUL, 2006

[3] www.wikipedia.com

[4] www.xilinx.com

[5] www.zilog.com/docs/z80/um0080.pdf